KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Business Information Technology

Jussi Piirainen

POWERSHELL TASK AUTOMATION FEATURES

Thesis
June 2018

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
June 2018
Degree programme in Business Information Technology

Karjalankatu 3
80220 JOENSUU
FINLAND
+358 13 260 600 (switchboard)

Author
Jussi Piirainen

Title
PowerShell Task Automation Features

Commissioned by
Valamis Group Oy

Abstract

The focus of this thesis is on researching and demonstrating PowerShell's capability as a task automation platform. The goal is to define a framework on which to implement, execute, manage and monitor automations and evaluate its capability and usability for automating tasks in the IT field. The fundamental task automation concepts and their corresponding features in PowerShell are introduced with practical use cases and examples.

The research focuses on PowerShell's native capability to identify the most suitable methods for automating tasks on any PowerShell instance. Practical use cases and implementations are introduced to demonstrate the capability of the researched features. Shorter examples are used to present the introduced features.

PowerShell offers the most essential features required for automating and managing tasks, but may require some development to avoid repeating similar steps in implementations. Some of the researched features are natively available in PowerShell, but rely on a separate Windows component, which limits their applicability on other operating systems.

Language
English

Pages 58

Keywords
PowerShell, automation, Windows, command line

Contents

1 Introduction

The goal of this thesis is to research and demonstrate PowerShell's capability as a task automation platform. This thesis concentrates on PowerShell's native capability in task automation. The focus is on the framework surrounding a task, rather than PowerShell's capability in automating a specific task. The goal is to identify essential features used in task automation to present a framework supporting automation implementation, execution, monitoring and management. PowerShell's task automation features are introduced with a relating concept, first explaining what the concept means in task automation, then moving to its practical use cases and applicable features in PowerShell. An evaluation of each feature's applicability and usability is also provided.

The introduced task automation features are demonstrated with some practical use cases and automation implementations, covering different kinds of common tasks in the IT automation area. The presented examples aim to take the introduced task automation feature into practice, explaining the used automation features in relation to the use case, also reasoning why each feature is used in the example. The goal of these examples is to prove the researched features' applicability in practice.

By reading this thesis, the reader should be able to choose the applicable task automation features for any given use case in the IT automation field. The introduced features are not meant to be solutions for a specific use case, instead they provide a framework on which to implement automations consistently and efficiently, regardless of the use case. The introduced features do not contain any customized functionality, everything is available in PowerShell either natively, although some of them rely on separate Windows or .NET components.

Chapter 2 begins with PowerShell's general description and version history. The basic functionalities and concepts are then introduced with brief descriptions. While usability is not a focus in this chapter, the basic concepts and features are introduced to better understand the following chapters. Chapter 3 covers the most

essential task automation concepts and features in PowerShell. The nature of the introduced methods varies, some are cmdlet-based features, while some combine scripting features or conventions into techniques usable in developing automations. Chapter 4 consists of some practical use cases and automation implementations. The features introduced in chapter 3 are illustrated in these practical examples by explaining and evaluating the applied features. Each example contains a brief description explaining the use case and a complete implementation developed for PowerShell. Chapter 5 concludes the thesis with the results.

2 PowerShell

2.1 Basics

Microsoft's PowerShell is a command line tool designed especially for IT system administrators and power users. PowerShell enables users to automate tasks requiring the operating system's functionality or installed applications. PowerShell was originally tied to Windows operating system, but has since widened its support to macOS, Linux and Unix based systems. (Microsoft 2017a.)

PowerShell improves upon traditional command line tools by offering a tool for building consistent and powerful automations. Consistency ensures that learning certain principles apply to the entire interface. Concepts such as naming conventions and feature discoverability in the tool itself apply themselves to every cmdlet. Just by developing a cmdlet, it is made discoverable and usable with a consistent syntax. The transition into scripting from inputting singular commands is designed to be as intuitive as possible. (Microsoft 2017a.)

PowerShell is built on top of Microsoft's .NET Framework (Windows PowerShell versions 1 – 5) or .NET Core (PowerShell Core, version 6). All output returned by PowerShell commands consists of .NET Framework objects. A major part of PowerShell's capability comes from .NET Framework's object-oriented approach, it is

PowerShell's main differentiating point compared to traditional shells. (Microsoft 2017b.)

2.2 PowerShell versions

During PowerShell's development, it was codenamed "Monad" with a shell called "msh". Before the first general availability release, Monad was renamed to Windows PowerShell (Microsoft 2006a). Windows PowerShell 1.0 reached general availability in November 2006 (Microsoft 2006b). The first release supported Windows XP SP2, Windows Server 2003 SP 1 and Windows Vista. 1.0 required .NET framework 2. (MacKechnie 2005.)

Version 2.0 was released along with Windows Management Framework (WMF) in October 2009. (Microsoft 2009.) Version 2.0 supported Windows XP, Windows Server 2003, Windows Vista and Windows Server 2008. Version 2 was also the first version which would be shipped as part of Windows Operating System (Windows Server 2008 R2 and Windows 7). (Microsoft 2008). The most important new features were the introduction of remoting, ISE (PowerShell Integrated Scripting Environment), background jobs (PSJobs) and modules. (Microsoft 2017c.)

Version 3.0 was released as part of WMF 3.0 in September 2012. (Microsoft 2012a.) Windows versions 7 SP1, Server 2008 R2 SP1 and Server 2008 SP2 were supported. Version 3 is also installed by default on Windows 8 and Windows Server 2012. (Microsoft 2017d.) This release included workflows, scheduled jobs, and remoting improvements. (Microsoft 2017e.)

Version 4.0 was released in October 2013 for Windows 7 SP, Windows Server 2008 R2 SP1. (Microsoft 2013a.) It is installed by default on Windows 8.1 and Windows Server 2012 R2. The most significant addition was the inclusion of Desired State Configuration (DSC). (Microsoft 2017f.)

Version 5.0 was released in February 2016 as part of Windows Management Framework 5.0 for Windows Server 2012, Windows Server 2008 R2 SP1, Windows 8.1 and Windows 7 SP 1. (Microsoft 2016a) Windows PowerShell 5.0 is installed by default on Windows Server 2016. The 5.0 release included class definitions, debugging features and DSC v2.0. (Microsoft 2017d.)

Version 6 was a major point in PowerShell's development. PowerShell was open sourced and renamed to PowerShell Core (previously Windows PowerShell). PowerShell Core utilizes .NET Core, the open sourced version of .NET, which allows it to support a wider range of operating systems. (Snover 2016). Operating systems supported by PowerShell Core 6.0 are listed in table 1.

Operating systems	**Versions**
Windows	7, 8.1, 10
Windows Server	2008 R2, 2012 R2, 2016
Ubuntu	14.04, 16.04, 17.04
Debian	8.7+, 9
CentOS	7
Red Hat Enterprise Linux	7
OpenSUSE	42.2
Fedora	25, 26
macOS	10.12+

Table 1: PowerShell Core 6.0 officially supported operating systems. (Microsoft 2018a).

Many of the new features in PowerShell Core 6 relate to its widened operating system support, including SSH support for remoting and renaming powershell.exe to pwsh.exe to support parallel installations of Windows PowerShell and PowerShell Core. Many minor changes were made to better support the shell-related conventions in Linux and macOS systems. (Microsoft 2018a).

At time of writing, PowerShell's latest version is 6.0.1. In this thesis, the shell is simply referred to as *PowerShell*. All cmdlets and features presented in this thesis

are valid for Windows PowerShell 5.1. Some of the presented features or cmdlets that rely on the Windows or the .NET Framework are no longer supported in PowerShell Core 6.0 due to its widened support for operating systems.

2.3 Commands

PowerShell commands can be categorized into three different types: cmdlets, functions and external commands. Cmdlets are PowerShell's native commands. They are written in PowerShell's scripting language or programming languages supported by the .NET Framework. (Microsoft 2017g.) PowerShell's native Cmdlets follow a specific naming convention consisting of a verb and a singular noun separated by a dash. This is also the recommended naming convention for any customized cmdlet. (Microsoft 2017h.) An example of this is shown in figure 1, where the `Write-Output` cmdlet is used to output text to the shell.

```
PS C:\ > Write-Output "Hello PowerShell"
Hello PowerShell
```

Figure 1: The "Hello World" of PowerShell. The `Write-Output` cmdlet is used to output text.

Functions are commands written in PowerShell's scripting language. Functions often consist of multiple commands, making reusing smaller script blocks easier. They fill the gap in between singular commands and reusable modules. It is recommended to follow the same naming conventions in functions as in cmdlets. (Jones, Hicks & Siddaway 2015, 34.)

External commands are executable files, such as .exe-files in Windows, traditionally ran from Windows command line (cmd.exe). External executables allow easy integration to external command line interfaces. (Jones, Hicks & Siddaway 2015, 34.)

Every available cmdlet has a help topic explaining a command's usage in detail. Help topics for functions and scripts are also supported. Cmdlet help topics can

be displayed by using the `Get-Help` cmdlet. For example, `Get-Help Get-ChildItem` returns a help topic for the `Get-ChildItem` cmdlet. Help topics contain extensive information about the cmdlet, including a list of all supported parameters. The `Get-Help` cmdlet also supports useful parameters like `-Examples`, which displays examples of using the cmdlet. (Microsoft 2017i.)

2.4 Parameters

Parameters enable users to customize the way commands are run. Unlike traditional shells, where commands have parameters with abbreviated names, PowerShell encourages usage of descriptive and consistent naming. (Microsoft 2017h). However, parameters do have a difference in naming convention compared to cmdlets, as parameters cannot contain spaces. This problem is bypassed by using “camel case”, where each word starts with a capital letter.

While many parameters are optional, some cmdlets require certain parameters to function. PowerShell will prompt the user for these mandatory parameters, if they are not specified with the cmdlet. Parameters can also have default values, which are used if the value is not explicitly set. Switch parameters are parameters that do not have values, instead they define some functionality to happen only if the parameter is provided with the cmdlet. (Jones 2012.) An example of a switch parameter is the `-Recurse` parameter of the `Get-ChildItem` cmdlet, stating that the results should include all files and directories including all recursively traversed subdirectories

Certain parameters, called “common parameters”, are available for all cmdlets. These parameters behave the same way, regardless of the command. An example of a common parameter is the `-Verbose` switch parameter, which outputs additional information during command execution. (Microsoft 2017h.)

2.5 Object pipeline

PowerShell commands are executed as part of a sequentially executed pipeline. Any output from a command can be passed on to the next command in the pipeline as input. Commands are chained into pipelines with the | character. Many commands require input values from the pipeline to match a certain object type. The object pipeline is an extremely important feature in PowerShell. The data returned by commands can be formatted, filtered and manipulated easily, without having to manually write multiple commands with iterate over many items. (Microsoft 2017j.) In figure 2, the `Get-ChildItem` cmdlet is first used as a singular command. In the second command the `Select-Object` cmdlet is piped after `Get-ChildItem` to limit the output properties to `Name` and `Length`.

```
PS C:\ps> Get-ChildItem
    Directory: C:\ps

Mode                 LastWriteTime         Length   Name
----                 -------------         ------   ----
d-----           2018-02-27 20:15                  script
-a----           2018-02-07 19:08              0   ps.ps1


PS C:\ps> Get-ChildItem | Select-Object Name, Length

Name     Length
----     ------
script
ps.ps1   0
```

Figure 2: An example demonstrating usage of PowerShell object pipeline.

In addition to manipulating the returned output, pipelines can be used to perform actions on the returned output. For example, using pipelines with the `Get-ChildItem` cmdlet allows users to perform many kinds functions on the resulting files or folders. The files could be moved (`Move-Item`) or copied (`Copy-Item`) to another folder with a single command placed in a pipeline after the `Get-ChildItem` cmdlet. (Microsoft 2017j.) An example of this is shown in figure 3, where the `Get-ChildItem` cmdlet is used to get all .png files by using the `-Filter` parameter (`*` is a wildcard, matching zero or more characters). The `Copy-Item` cmdlet is placed in a pipeline, copying all resulting .png files to the destination directory.

```
PS C:\ > Get-ChildItem -Filter *.png | Copy-Item -Destination C:\pictures
```

Figure 3: The `Get-ChildItem` cmdlet is used to copy image files to another directory.

2.6 Variables

PowerShell variables are used to temporarily store data in a PowerShell session. Any output from a command can be placed into a variable, to be used later rather than running a command again to get the same results (Jones, Hicks & Siddaway 2015, 260). Variable values can also be set manually. Variables follow the object typing rules set by .NET Framework. As all output returned by cmdlets have object types, variables save the object, rather than the text-based representation of it, which is written to the console window. A variable must have a name, which cannot contain spaces, all alphanumeric characters and underscores are allowed. (Microsoft 2017k.) A variable is created by using the `$` (dollar) character followed by a valid variable name. Variables names should be descriptive and consistent, especially in scripts. Variable names are case-insensitive, camel-case can be used to make variable names more readable (Microsoft 2017l). In figure 4, the result of the `Get-Date` cmdlet is placed into the `$date` variable. Typing a variable's name into PowerShell outputs its value. The `Get-Date` cmdlet returns the current date on the local computer. The object type of `$date` is `DateTime`.

```
PS C:\> $date = Get-Date
PS C:\> $date

Wednesday, May 9, 2018 18:54:15
```

Figure 4: A simple example of using a variable in PowerShell.

By default, a variable can hold any type of object. It is possible to define the type of a variable, making it impossible to set its value to another type of an object. In some cases, this can be confusing as PowerShell automatically attempts to con-

vert values to an object type corresponding the variable. (Jones, Hicks & Siddaway 2015, 263.) The type of a variable can be declared by specifying `[<type>]` before the variable name, for example `[int]$number` would declare that the variable `$number` can only contain integer values. An example of this behaviour is shown in figure 5.

```
PS C:\> [int]$number = 8
PS C:\> $number = "5"
PS C:\> $number = Get-Date

Cannot convert value "10.5.2018 10.48.57" to type "System.Int32". Error: "Invalid
cast from 'DateTime' to 'Int32'."
At line:1 char:1
+ $number = Get-Date
+ ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Figure 5: An error produced by an invalid conversion attempt.

In figure 5, the type of `$number` is set to integer (`[int]`). Placing a string value “5” into an integer type variable does not produce an error, because PowerShell automatically converts the value to integer. However, the resulting object from `Get-Date` is of type `DateTime`, which cannot be placed into the `$number` variable. Attempting to do so will cause an error.

Variables can be used as parameter values, to make it easier to utilize results of commands. Variables can also be used in the scripting language of PowerShell. Variable values can be compared to create conditional processing logic or used with loops to manipulate values in a collection. (Jones, Hicks & Siddaway 2015, 263.) In figure 6, a new file is created by using a value stored in a variable.

```
PS C:\> $filePath = "C:\PowerShell\Test.txt"
PS C:\> New-Item -Path $filePath


    Directory: C:\PowerShell

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        10.5.2018     11.32              0 Test.txt
```

Figure 6: A file path is stored in the `$filePath` variable. The `New-Item` cmdlet is used to create a new file to the path specified by the `$filePath` variable.

Some variables are created automatically by PowerShell. Automatically created variables store data about the state of the PowerShell session. Their values cannot be set by the user, PowerShell updates their values as needed. (Microsoft 2017m.) For example, the variable `$PSVersionTable` contains PowerShell's detailed version information.

2.7 Scripts and functions

PowerShell provides a scripting language for creating more complex automations. The scripting language is similar to C#, the .NET programming language, supporting basic programming language features, designed specifically for a shell-based environment. (Jones, Hicks & Siddaway 2015, 317.) Scripting allows combining multiple commands into singular parametrized scripts or functions, providing features such as conditional execution, looping, variables and arithmetic operations. (Jones, Hicks & Siddaway 2015, 61.)

Multiple sequentially executed commands can be placed into a function or script to make the functionality available by typing a single command. Functions and scripts can be distributed and executed from script files, which have a .ps1 file extension. Scripts can be executed by inputting a full file path to the .ps1 file or with the dot sourcing operator "`.`" (period). When the dot sourcing operator is used, the script is run in the current PowerShell session scope, making functions defined inside the script available until the session is closed. (Microsoft 2017m) When a script file is executed, it's code is run as-is in the current session. The difference between a script and a function is that functions can remain (with dot sourcing) available in the interactive session as commands. (Microsoft 2017n.) In figure 7, the `TestScript.ps1` script is executed in a PowerShell session, making the function available to use until the session is closed. Note the dot sourcing operator used to execute the script.

```
# C:\powershell\TestScript.ps1
Function Get-CurrentTime
{
    $date = Get-Date
    $date.ToString("HH\:mm\:ss\.fff")
}


---


PS C:\> . C:\powershell\TestScript.ps1
PS C:\> Get-CurrentTime
12:07:43.880
```

Figure 7: The `TestScript.ps1` script file contains the `Get-CurrentTime` function, which outputs current time.

Scripts and functions can define parameters to be used as input values similarly to cmdlets having parameters. Parameters are defined with the same syntax for functions and scripts. Parameter definitions can be placed in a `Param()` block. The definitions are similar to variable definitions, containing the variable name with the $ sign and optionally the object type in brackets. There are also additional options to control how the parameter is handled. (Microsoft 2012b.) For example, a parameter can be defined to be mandatory as shown in figure 8. Not providing a mandatory parameter will produce an error.

```
# C:\powershell\Get-Time.ps1
Param(
    [Parameter(Mandatory=$True)]
    [DateTime]$Date
)

$date.ToString("HH\:mm\:ss\.fff")


---


PS C:\> .\Get-Time.Ps1 -Date (Get-Date)
12:51:12.795
```

Figure 8: The `Get-Time.ps1` script returns a formatted date. The `-Date` parameter of `Get-Time.ps1` script is mandatory.

2.8 Modules

PowerShell's capability can be extended by using modules. Modules enable developers and software vendors to easily integrate into PowerShell or extend its native capability by offering customized cmdlets. Related functionalities can be packaged as a module to be easily shared and installed. Modules consist of code, dependencies and manifest files. Modules can be divided into two types: script modules and binary modules. Script modules contain any valid PowerShell code. Binary modules contain compiled code written in a .NET programming language such as C#. Manifest files can optionally be used to store related metadata, such as versioning, dependencies and author information. (Microsoft 2018b).

PowerShell modules can contain scripts or cmdlet definitions, which are made available either by importing the module manually by using the `Import-Module` cmdlet or by using the autoloading feature, which imports modules placed into directories defined in the `PSModulePath` environment variable. Whenever a cmdlet in `PSModulePath` directory is called, PowerShell automatically loads the module into the session and executes the requested command. (Microsoft 2018c).

PowerShell also supports PSSnapins, an older method for extending functionality within PowerShell. However, they PSSnapins are not covered in these thesis, since PowerShell's version 2, modules have been the preferred way of creating customized cmdlets. (Microsoft 2018d).

Even though Microsoft ships certain products with PowerShell-powered management shells, it should be noted that these shells are simply PowerShell instances with automatically loaded modules, not product-specific versions of PowerShell. The customizations made in these consoles can be replicated in PowerShell by examining the properties of said instances. This allows users to combine many preloaded extensions into one powerful PowerShell instance. (Jones, Hicks & Siddaway 2015, 44).

3 Task automation features in PowerShell

3.1 Introduction to task automation

Although the automated tasks can be very different from each other, certain principles hold their ground among most tasks. The most basic idea of automation is to perform a task more efficiently than a human would, while maintaining a higher level of consistency and quality. Generally, it is more beneficial to aim for as independent automation as possible, keeping manual configuring or user interaction to a minimum. However, when tasks are automated, ways must be implemented to monitor these automations, which makes the automation development process longer. In many cases, independent execution is made possible by following an execution schedule or a trigger, defining when a task should be executed.

Task automation reduces human errors and makes the actual execution of a task visible. When automations are executed, tasks are performed in order, the same way every time. A human is no longer needed to actively monitor what is being done, instead the automation generates execution logs, describing each step in the automation. Execution logs can be examined to find out exactly what happens during a task. Automations can be used to document previously undocumented tasks, making it possible to examine them afterwards or to produce and analyse accurate data from the processes.

This chapter introduces PowerShell's task automation features and concepts. While all of PowerShell's features relate in some way to task automation, this chapter aims to introduce the most essential task automation concepts and their applicable features in PowerShell as well as introduce some surrounding techniques. The presented features allow users to control how, where and when tasks are executed.

All introduced features and concepts contain a description of how and why they should be used. Deductions on where the methods could be usable are also

made. Using all the introduced task automation features without exception is not intended as the use case defines the suitable methods and features. Being able to recognize and apply the right automation methods requires one to understand their capability and benefits. Combining the introduced features correctly creates a framework on which to build consistent and robust automations.

3.2 Task

A task in PowerShell can be understood as any executable command, be it external commands, cmdlets, script blocks, functions, scripts or workflows. Most of the time tasks are complex enough so that singular commands will not be sufficient for the automation implementation. Most tasks require more complex logic or simply more lines of commands, which is why command wrappers, such as scripts or functions, are more common in automation implementations.

A script or a function can define a generic structure for a specific use case. Parameters are used to make generic structures applicable for multiple use cases. The parametrized values are often environment or subject-specific information such as server names or file paths. Many parts of automations can be directly reused by using PowerShell's modules and scripts, although some conventions should be followed in development to ensure unified management and maintenance. In many cases, functions can be moved into modules and then used in automation implementations as a way of enforcing certain conventions and avoiding redundancy in development.

3.3 Logging

Traditionally as the user is working in a command line environment, the steps of a task are produced as the user inputs commands into the shell. The user rarely writes down descriptive comments of what is happening. Instead the user can probably describe verbally what was done and what errors occurred. As automa-

tions are implemented, multiple commands are executed very quickly, each executing right after the previous one completes. If no logging is implemented, it can be difficult to identify exactly where an error occurred, or what branch of a conditional logic was followed. It is important that the steps taken to complete a task in an automation remain visible and auditable when a task is automated.

In PowerShell, logging can be implemented by using the redirection operators, which redirect various messages sent by PowerShell to a text file. The general syntax for using these operators is to specify any command followed by the `>` or `>>` operators and a file path. The `>` operator states that all output of a command is appended to a new text file, overwriting the target file if it already exists. The `>>` operator appends output to an existing file. When output is redirected, it is not shown on the session host window. The example in figure 9 uses the `>` operator to send output to a text file. The redirected output is then retrieved with `Get-Content`.

```
PS C:\> Get-Date > "log.txt"
PS C:\> Get-Content -Path "log.txt"

Saturday, May 12, 2018 11:26:07
```

Figure 9: The `>` operator is used redirect the output of `Get-Date` to a text file.

The `>` and `>>` operators only redirect success output to text files. Successful output, however, is rarely of use in finding out why errors happened. PowerShell handles output by sending different types of output messages to corresponding message streams. The message streams define how each type of message is processed (Blender 2014). For example, the success output stream is used to send messages further into the PowerShell object pipeline. Messages from other streams are ignored by the pipeline, unless they are specifically redirected to the success output stream. The streams which process all kinds of output are shown in table 2.

Stream	**Description**
(1) Success	Any successful output sent by cmdlets.

(2) Error	Errors produced by commands or manually written with the Write-Error cmdlet.
(3) Warning	Warnings written with the Write-Warning cmdlet.
(4) Verbose	Verbose messages written with the Write-Verbose cmdlet.
(5) Debug	Debug messages written with the Write-Debug cmdlet.

Table 2: The PowerShell streams used to handle any messages sent by the PowerShell engine (Blender 2014).

The relevancy of streams in relation to logging comes from controlling which messages are redirected to file. The usage of the `>` and `>>` operators can be extended by specifying a stream from which the messages are redirected. This is done by prepending the number identifier of the stream (the number in parentheses in table 2) to the redirection operators. For example, the `4>` operator redirects only verbose messages to file. In addition to specifying a single stream, the asterisk (*) character can be used to represent all streams, writing output from all streams to file. The verbose and debug streams are not displayed by default, they can be enabled by specifying the `-Verbose` and `-Debug` common parameters on any cmdlet (Microsoft 2018e).

When implementing logging to a task, it is important to follow some conventions to avoid creating pitfalls or complications when the task is executed by someone not aware of the implementation. When streams are used correctly, all tasks can behave the same way, regardless of the use case. The strict rule is that no task should use the success output stream to describe the execution, even though it is very easy to use `Write-Output` as a cmdlet for logging for example execution progress. The success output should only be used to produce output to the object pipeline. Similarly, the error stream should only be used for errors, leaving the user to decide how the errors are handled. The warning stream should be used to display messages about something not going entirely as expected. The verbose stream is the stream that should be used to describe execution, at critical

points of action. The debug stream is used as part of the debugging features in PowerShell (Microsoft 2018f).

The most essential messages to log, success and error output, are displayed and sent to corresponding streams automatically. In many cases, especially in simpler tasks, it is enough to let PowerShell automatically produce the output and errors it naturally produces, without explicitly defining descriptive verbose messages. However, when tasks have high amounts of actions, it can be useful to describe the process verbosely. In practice, the `Write-Verbose` cmdlet is used to write descriptive messages relating to the state of execution, usually at the most critical actions within the task. When a user executes the task, the verbose logging can be turned off or on, depending on the user's preference, either by specifying the `-Verbose` common parameter or using the `$VerbosePreference` variable (Microsoft 2018g).

```
# Compress-Files.ps1

[CmdletBinding()]
Param(
    [Parameter(Mandatory=$true)]
    [string]$Directory,
    [Parameter(Mandatory=$true)]
    [string]$Destination
)

Write-Verbose "Looking for files in $Directory"
$files = Get-ChildItem -Path $Directory -File | Select-Object -ExpandProperty
"FullName"
Write-Verbose "Zipping $($files.Count) files to $Destination"
Compress-Archive -Path $files -DestinationPath $Destination


---


PS C:\powershell> ./Compress-Files.ps1 -Destination "C:\powershell\backup.zip" -
Directory "C:\powershell\log" -Verbose *>> backuplog.txt
PS C:\powershell> Get-Content backuplog.txt

VERBOSE: Looking for files in C:\powershell\log
VERBOSE: Zipping 4 files to C:\powershell\backup.zip
```

Figure 10: An example of using `Write-Verbose` in a script.

In the script `Compress-Files.ps1` shown in figure 10, The `Compress-Files.ps1` script has verbose messages defined to describe the state of its execution. The

script is called using the `-Verbose` common parameter, enabling the verbose messages. All messages from streams 1 to 4 are redirected to a log file.

The `[CmdletBinding()]` attribute definition (included also in figure 10) states that the script behaves like a compiled cmdlet, enabling among other features the usage of common parameters, in this case the `-Verbose` switch parameter. The `Param()` block is used to define the parameters used by the function. Common parameters, including the `-Verbose` switch, are automatically available on scripts or functions that contain the `[CmdletBinding()]` attribute or at least one `[Parameter()]` definition. (Microsoft 2017o.)

In addition to the redirection operators, PowerShell has support for transcripts, which are used for logging any operations done in a PowerShell session to a text file. Transcripts log all output displayed on the host window, as well as additional information about the session. A transcript is started with the `Start-Transcript` cmdlet. By default, PowerShell will name the transcript, optionally the `-Path` parameter can be used to specify a destination file path. If the optional `-Append` parameter is not used, the destination file is overwritten. (Microsoft 2017p.)

```
PS C:\powershell> Start-Transcript -Path "C:\powershell\history.log"
Transcript started, output file is C:\powershell\history.log
PS C:\powershell> ./Compress-Files.ps1 -Destination "C:\powershell\backup2.zip" -
Directory "C:\powershell\log" -Verbose
VERBOSE: Looking for files in C:\powershell\log
VERBOSE: Zipping 4 files to C:\powershell\backup2.zip
PS C:\powershell> Stop-Transcript
Transcript stopped, output file is C:\powershell\history.log
```

Figure 11: The transcript feature is used to log the output of `Compress-Files` script featured in figure 10.

As demonstrated in figure 11, the biggest difference in using transcripts compared to redirection operators is that the output is still displayed in the console host window. The biggest benefit of using the transcripts is the additional information which is logged at the start of a transcript. This information can be valuable when tracing and debugging errors. All the additional information provided by the transcript is shown in figure 12.

```
**********************
Windows PowerShell transcript start
Start time: 20180512193742
Username: WLAB\Administrator
RunAs User: WLAB\Administrator
Machine: WSERVER7 (Microsoft Windows NT 10.0.14393.0)
Host Application: powershell
Process ID: 3728
PSVersion: 5.1.14393.1884
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.14393.1884
BuildVersion: 10.0.14393.1884
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
```

Figure 12: The additional information at the beginning of a transcript, which is only written to the destination file, instead of the session host window.

The two introduced methods, redirection operators and transcripts, provide a way to easily add logging to an existing task. PowerShell makes it easy to leave the logging up to the user. Adding logging to any command, including scripts and functions, is extremely easy with redirection operators. Transcripts require adding some commands to start and stop the transcript, requiring a bit more effort. In both cases, the logging should not be forced in the actual automation. If additional logic is needed for the logging, for example generating a name for the log file, the logic should be implemented without interfering with the actual task. This could be done by wrapping a script in another script containing the logic specific for a specific task.

The decision of when to use redirection or transcripts is driven by the use case. Transcripts should be used if the additional information logged by transcripts can be seen valuable for debugging or auditing purposes. For example a task's target computers have different PowerShell versions or user permissions, the information might be valuable for debugging. Transcripts are also useful when the user works in a single interactive PowerShell session, inserting varying commands. A transcript can be started, leaving all actions visible afterwards. Redirection operators can implement logging very quickly, the biggest difference from transcripts being the ability to limit the number of messages by specifying a stream to redirect.

3.4 Remoting

In task automation, one of the main goals is to save time. Often the tasks that are repetitive generate the biggest amount of work. Often the need to repeat a task comes from completing it on multiple computers. PowerShell remoting features enable tasks to execute on remote computers, removing the need to complete steps to access a remote computer manually before executing an automation. Remoting also allows centralizing the management of automations to a single point of access. This chapter focuses on cmdlets which are specially meant for accessing remote computers or executing commands on them. While some cmdlets have built-in functionality to allow remote execution, most cmdlets have no such functionality. PowerShell remoting allows users to run any command on a remote computer. PowerShell remoting requires certain requirements to be met in network configurations and user permissions (Microsoft 2018h).

Interactive PowerShell sessions can be started and accessed remotely. The remote PowerShell session functions as it would if the user was accessing the computer locally. The `New-PSSession` cmdlet can be used to create a new remote session on a remote computer. The remote computer name is specified with the `-ComputerName` parameter. The remote session can be entered with the `Enter-PSSession cmdlet`, by specifying either the session name (`-Name` parameter) or the session ID (`-ID` parameter). The `Enter-PSSession` can also be used to create and enter a remote session by using the `-ComputerName` parameter. When a remote session is entered, the beginning of the shell prompt will specify the remote computer name in brackets as shown in figure 13.

```
PS C:\Users\Administrator.WLAB> New-PSSession -ComputerName WServer5

 Id  Name     ComputerName ComputerType  State  ConfigurationName    Availability
 --  ----     ------------ ------------  -----  -----------------    ------------
  4  Session4 WServer5     RemoteMachine Opened Microsoft.PowerShell    Available


PS C:\> Enter-PSSession -Id 4
[WServer5]: PS C:\> Exit-PSSession
PS C:\>
```

Figure 13: A new remote session is created with `New-PSSession`. The output contains the session ID, which is used to enter the session with `Enter-PSSession`.

In the remote session any cmdlet can be executed on the remote computer, as long as the command is available on the remote computer. The `Exit-PSSession` cmdlet (also present in figure 13) can be used to exit a remote session. Exiting a remote session will not remove the remote session, the session will remain available until it is removed with the `Remove-PSSession` cmdlet.

Another way to execute commands on the remote computer is to use the `Invoke-Command` cmdlet. The `Invoke-Command` cmdlet can execute any number of commands on a remote computer. The commands are placed in a script block, specified with the `-ScriptBlock` parameter. As with remote sessions, the remote computer is specified with the `-ComputerName` parameter. An example of running a command on a remote computer is demonstrated in figure 14.

```
PS C:\> Invoke-Command -ComputerName "WServer6" -ScriptBlock { Restart-Service
"VaultSvc" }
```

Figure 14: `Invoke-Command` is used to restart a service on a remote computer. Successfully restarting a service produces no output.

`Invoke-Command` can run the same script block on multiple machines simply by defining more computer names in the `-ComputerName` parameter. This makes it extremely easy to execute a task on multiple remote computers in parallel. Virtually any command can be executed on a remote computer with the `Invoke-Command` cmdlet, even script files from the local computer can be sent from the local machine to be executed on the remote machine. Script files can be specified with the `-FilePath` parameter of `Invoke-Command`.

The output from remote execution initiated with the `Invoke-Command` cmdlet is returned to the local PowerShell session, displayed as any other output. The outputs from target computers are automatically combined into a single collection. The `PSComputerName` property is added to tell the user where the returned data

was generated. (Microsoft 2017q.) This is especially useful when multiple target computers are used. In figure 15, `Invoke-Command` is used to gather IP addresses from remote computers. The script block assigned to the `$commands` variable gets the IP address of the local computer. The script block is executed on three remote computers with `Invoke-Command`. The `PSComputerName` property is added to the output, specifying which computer returned the result.

```
PS C:\> $commands = { Get-NetIPAddress | Where-Object -Property IPAddress -like
"192.168.1*" | Select IPAddress }
PS C:\> Invoke-Command -ScriptBlock $commands -ComputerName
"WServer5","WServer6","WServer7"

IPAddress       PSComputerName   RunspaceId
---------       --------------   ----------
192.168.1.101   WServer5         9da3d16e-3394-47d0-9bba-5b796f74f461
192.168.1.103   WServer6         d0a99d27-9a11-4478-9ca6-91574eb1e138
192.168.1.107   WServer7         100662de-85e8-42d8-816b-05dcfb3d5ef5
```

Figure 15: Invoke-Command is used to retrieve IP addresses of remote computers.

The remote session features mainly provide a way for users to work directly in a remote session, entering commands one by one, so the use for independent automation is not very high. `Invoke-Command` however allows rapid distribution of tasks that are either created by quickly specifying small amounts of commands into script blocks or by executing scripts, which might contain more complex automations (Microsoft 2018i). The effort required to scale the automation to multiple computers only requires the task to be suitable for execution on the target computers, specifying the target computers is trivial.

The remoting features do not necessarily serve any specific use case. Instead they provide an additional framework on which to execute tasks. Implementing remoting manually into tasks would be difficult or at least time intensive. If a command or a script runs on the local machine, it is very probable that it will run on a remote computer, as long as the environments are relatively similar. Overall the remoting features, once configured, are very intuitive to use, allowing remoting to be easily integrated into automations.

3.5 Background jobs

Tasks that do not require user interaction can be run asynchronously in the background without a dedicated interactive shell assigned to them. This is useful when a task execution requires a lot of time or when multiple tasks are executed in parallel. When tasks are executed in the background, the user can interact with the shell without interruption, completing other tasks as needed (Microsoft 2017r). The running task's status can be monitored, and when it completes, it's output can be retrieved easily. Tasks can for example be run on multiple machines and once all tasks are completed the output data can be combined to create a dataset representing an entire environment.

In PowerShell, a managed task ran in the background is called a background job. Support for executing and managing background jobs is implemented in PowerShell with a set of related cmdlets which allow starting, monitoring and retrieving results of jobs. A background job is started with the `Start-Job` cmdlet, which executes a script block as a background task. A script block is a piece of PowerShell code, defining the commands and logic to run, wrapped in curly brackets. (Microsoft 2017r.) A script block can be used to quickly define a task to be run on demand. Larger tasks can be placed into script files, and then be invoked from within the script block or by using the `-FilePath` parameter instead of `-ScriptBlock`. An example of using the `Start-Job` cmdlet in presented in figure 16, where a script block is executed as a background job. The command placed in the script block looks for `ServerSpec.txt` file in the C drive and its subdirectories.

```
PS C:\> Start-Job -ScriptBlock {Get-ChildItem -Path "C:\" -Filter "ServerSpec.txt"
-Recurse -ErrorAction "SilentlyContinue"}

Id     Name     PSJobTypeName   State      HasMoreData     Location          Command
--     ----     -------------   -----      -----------     --------          ------
1      Job1     BackgroundJob   Running    True            localhost         Get...
```

Figure 16: A script block is started as a background job with the `Start-Job` cmdlet.

PowerShell automatically names the started job, if a name is not specified with the `-Name` parameter. Specifying descriptive names for jobs is a good convention, as it makes recognizing certain tasks quicker, especially when multiple tasks are run from a session. In addition to outputting the job name, `Start-Job` returns the unique identifier in the Id column immediately after running the command. It is important to take note job ID's and names, as they can be used later to retrieve the job's state or results

Background jobs are monitored with the `Get-Job` cmdlet, which yields information about jobs. Jobs created by the `Start-Job` cmdlet are only visible until the current session is closed. In figure 3, the output of `Get-Job` contains 3 jobs that have been started from the session, their Id's, names and states are formatted into a table, as shown in figure 17.

```
PS C:\> Get-Job

Id  Name  PSJobTypeName   State      HasMoreData    Location    Command
--  ----  -------------   ---------  --------       -------     -----
1   Job1  BackgroundJob   Completed  False          localhost   Get-Process | Wher...
3   Job3  BackgroundJob   Running    False          localhost   Get-ChildItem | Wh...
5   Job5  BackgroundJob   Failed     True           localhost   Invoke-RestMethod ...
```

Figure 17: The `Get-Job` cmdlet is used to retrieve information about jobs started from the current PowerShell session.

The output of a completed background job is retrieved with the `Receive-Job` cmdlet, regardless of it's state. By default, after running `Receive-Job` on a job, the output is removed and can no longer be retrieved. The results can be preserved with the `-Keep` switch of `Receive-Job`. If a job is running when it's data is retrieved, only the output gathered so far will be included. Output data can be gradually retrieved until the job is fully completed. Figure 18 presents an example of starting a background job and retrieving its output with the `Receive-Job` cmdlet, by referencing the `Id` property output by the `Start-Job` cmdlet.

```
PS C:\> Start-Job -ScriptBlock { Get-ComputerInfo | Select
WindowsInstallDateFromRegistry, TimeZone}

Id   Name   PSJobTypeName  State    HasMoreData  Location   Command
--   ----   -------------  -----    -----------  --------   -------
17   Job17  BackgroundJob  Running  True         localhost  Get-ComputerInfo | Se...
```

```
PS C:\> Receive-Job 17

WindowsInstallDateFromRegistry : 2017-12-01 05:59:57
TimeZone                       : (UTC+02:00) Helsinki, Kyiv, Riga, Sofia, Tallinn,
Vilnius
RunspaceId                     : 03c5f003-6c7d-4427-bf23-87de78e221e9
```

Figure 18: A background job fetches the local computer's time zone and Windows install time.

The real time save in using background jobs comes from the the ability to run them on remote computers. Background jobs can be started on a remote computer by using the `Invoke-Command` cmdlet with the `-AsJob` switch parameter. The `-AsJob` parameter invokes the specified command or script block as a background job, executing the job on a remote machine, but allowing managing the task on the local computer. The results of background jobs executed on remote computers are returned to the local computer. (Jones, Hicks & Siddaway 2015, 183.) The job is only tracked on the local computer. As with `Invoke-Command` generally, background jobs ca also be started on multiple target computers. The specified command or script block is executed simultaneously on all target computers, the results are returned to the local computer as soon as they complete. (Microsoft 2017s.) The target remote computer or computers are defined with the `-ComputerName` parameter. In figure 19 a script block is executed on a set of remote computers. Output from the remote jobs is retrieved with a single command, which outputs results from all target computers.

```
PS C:\> Invoke-Command -ScriptBlock { [PSCustomObject]@{"Free" = (Get-PSDrive -Name C
| Select Free).Free/1GB} } -ComputerName WServer5,WServer6,WServer7 -AsJob

Id   Name    PSJobTypeName  State    HasMoreData  Location             Command
--   ----    -------------  -----    -----------  --------             -------
25   Job25   RemoteJob      Running  True         WServer5,WServer6... [PSCustomOb...


PS C:\> Receive-Job 25 | Select-Object PSComputerName,Free

PSComputerName             Free
--------------             ----
WServer5       12.0516929626465
WServer6       14.4823799133301
WServer7       15.3433271142578
```

Figure 19: `Invoke-Command` is used to collect information about hard disk space on three (WServer5, WServer6 and WServer7) remote computers. The available space is output for each remote computer.

Background jobs that run on multiple target computers automatically generate hidden child jobs for each remote machine, linked to a single parent job. These child jobs can be managed separately. The child jobs can be included in the output of `Get-Job` by using the `-IncludeChildJob` switch parameter. While the child jobs only control and monitor the state of a job on one remote computer, the parent job's state is failed if just one of the child jobs fails. (Jones, Hicks & Siddaway 2015, 184.)

Background jobs offer a flexible way to run tasks interactively, maximizing the productivity in working from within a single interactive shell. The cmdlets related to management of background jobs are very intuitive, they only require the user to run commands or script files using the `Start-Job` or `Invoke-Command` cmdlets. Most of the time the commands or scripts do not even require special care to be compatible with background jobs. The time saving benefits are seen especially when running commands on multiple machines, where the task execution time will follow the worst-case scenario. Varying amounts of data between computers can lead to extended execution times on some computers. While the execution may have completed on most of target machines, the task is not completed until the slowest execution is finished. By using background jobs, partial results can be used, or other tasks can be completed while the task is running.

Another great benefit of background jobs is that it makes managing jobs and their results easier. Normally a user must manage the tasks and their results manually, saving data into variables or reviewing output history of previous commands. Background jobs make management of task and their results easier. A user can review past tasks and retrieve results of said tasks at any time.

3.6 Scheduled jobs

Background jobs by themselves only allow starting tasks on demand from an interactive PowerShell session. While this allows the user to work on other tasks while the tasks are running, the process in its entirety still requires user interaction. Tasks can often be periodically recurring, in which case schedules can be utilized to trigger the jobs. Scheduling bypasses the limitation of working hours, allowing tasks to run on servers any time, depending on the nature of the task, preferably when the overall load is lower.

When a scheduled task triggers, all information needed to execute the task must be made available beforehand. The information is gathered from within a task, using predefined values, filesystems, API's or databases. The user cannot be prompted for information during execution. Error handling raises its importance in scheduled automations as the user is not there to immediately react to errors. Tasks which utilize scheduled automation should generally be made more error resistant, which requires a bit more effort in development, but results in more robust automations.

PowerShell's scheduled jobs extend background jobs' functionality by adding scheduling capabilities and job persistence through shell sessions (Microsoft 2017t). Scheduled jobs are managed using the same cmdlets as with background jobs in addition to a set of cmdlets specifically designed for managing scheduled jobs. The cmdlets used for scheduled jobs are contained in the PSScheduledJob module, which is installed by default in PowerShell. PSScheduledJob module relies on the Windows Task Scheduler to provide support for scheduling features. (Jones, Hicks & Siddaway 2015, 189.) Scheduled jobs are saved on the local computer, but they can be managed remotely by using PowerShell's remoting features.

3.6.1 Triggers

A scheduled job requires a trigger to define when the job is run. Triggers are created with the `New-JobTrigger` cmdlet, which merely creates a trigger. A trigger can can then be attached to scheduled job. Triggers support many kinds of parameters to customize periodic, one-time or simple event-based triggering (At logon and at system startup). For example, a trigger for daily tasks can be created using the `-Daily` switch parameter with the `-At` parameter to define a time of day. Table 3 contains some examples and descriptions of triggers.

Trigger definition	**Description**
`New-JobTrigger -Daily -At "18:50"`	Every day, at 18:50.
`New-JobTrigger -Weekly -DaysOfWeek Tuesday,Saturday -At "03:45" -WeeksInterval 3`	Every three weeks on Tuesday and Saturday at 03:45.
`New-JobTrigger -AtLogOn`	At user logon.
`New-JobTrigger -AtStartup`	At system startup.
`New-JobTrigger -At 21:00 -Once -RepeatIndefinitely -RepetitionInterval (New-TimeSpan -Hours 1)`	Repeat indefinitely every 1 hour starting at 21:00.

Table 3: Examples of scheduled job triggers.

One scheduled job can have multiple triggers. Triggers can be attached to the job when the job is created or added later with the `Add-JobTrigger` cmdlet. Triggers can be managed separately from the jobs they are attached to, but do not persist if they are removed from a scheduled job. Existing triggers can be disabled, enabled or edited as needed. The cmdlets related to scheduled job triggers are listed in table 4, with short descriptions of the cmdlets.

Cmdlet	**Description**
`New-JobTrigger`	Creates a new trigger.
`Add-JobTrigger`	Attaches a trigger to a scheduled job.
`Remove-JobTrigger`	Removes an attached trigger from a scheduled job.

`Get-JobTrigger`	Gets triggers attached to a scheduled job.
`Set-JobTrigger`	Changes the properties of an existing trigger.
`Disable-JobTrigger`	Disables a trigger (without removing it).
`Enable-JobTrigger`	Enables a trigger.

Table 4: The cmdlets used in creating and managing scheduled job triggers (Jones, Hicks & Siddaway 2015, 191.).

3.6.2 Managing scheduled jobs

A scheduled job can be created with the `Register-ScheduledJob` cmdlet. As with the `Start-Job` cmdlet, the task is defined either with a script block or by defining a script file to run. Unlike in background jobs, scheduled jobs need to be named with the `-Name` parameter. Triggers can be created in the `-Trigger` parameter or by creating the trigger beforehand, in which case the trigger needs to be assigned to a variable. If a user account is not defined with the `-Credential` parameter, the scheduled job is run on the current user. A list of scheduled jobs and their states can be retrieved with the `Get-ScheduledJob` cmdlet. Scheduled jobs are deleted with the `Unregister-Scheduled` cmdlet, all job results from the scheduled job are also removed (Microsoft 2018j). Figure 20 shows an example of creating a scheduled job by first creating the trigger and assigning it to a variable. The trigger is then attached to a scheduled job as it's created with the `Register-ScheduledJob` cmdlet. The job removes files older than 7 days in `C:\log` directory. The job is scheduled to run at 07:00 every Sunday.

```
PS C:\> $trigger = New-JobTrigger -At 07:00 -Weekly -DaysOfWeek Sunday

PS C:\> Register-ScheduledJob -Name RemoveLogFiles -ScriptBlock { Get-ChildItem -File -Path C:\log
| Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-7) } | Remove-Item } -Trigger $trigger
```

Figure 20: Creation of a scheduled job using the `Register-ScheduledJob` cmdlet.

Scheduled jobs have several options, which are assigned default values if they are not explicitly set at time of creation. Some of these options are essential to task automation, providing such features as controlling if a job is run without network connectivity (`-RequireNetwork`) or running the task with elevated permissions (`-RunElevated`). Scheduled job options can be set by first creating a ScheduledJobOptions object with the `New-ScheduledJobOption` cmdlet. The ScheduledJobOptions object can then be attached to a scheduled job with the `-ScheduledJobOption` parameter of `Register-ScheduledJob` or `Set-ScheduledJob` cmdlets. The `Set-ScheduledJob` cmdlet will overwrite values of existing options. The functionality of the `Get-ScheduledJob` is demonstrated in figure 21, where the existing scheduled job created in figure 20 is set to wait for the computer to be idle for 10 minutes before running the task, for a maximum of 2 hours, after which the task is not run at all.

```
PS C:\> $opt = New-ScheduledJobOption -IdleTimeOut (New-TimeSpan -Hours 2) -
IdleDuration (New-TimeSpan -Minutes 10)

PS C:\> Get-ScheduledJob -Name "RemoveLogFiles" | Set-ScheduledJob -
ScheduledJobOption $opt
```

Figure 21: An example of editing an existing scheduled job's options.

The cmdlets used with background jobs can also be used to start scheduled jobs and retrieve their results. Scheduled jobs can be run on demand by using `Start-Job` with the `-DefinitionName` parameter, specifying the scheduled job name. When scheduled jobs are started with the `Start-Job` cmdlet, they are treated as background jobs, so their results will not persist automatically through PowerShell sessions. Jobs triggered by scheduled jobs are viewed with the `Get-Job` cmdlet. Background jobs and scheduled jobs can be distinguished from each other with the `PSJobTypeName` property as shown in figure 8.

```
PS C:\> Get-Job

Id   Name            PSJobTypeName   State      HasMoreData Location    Command
--   ----            -------------   -----      ----------- --------    -------
13   RemoveLogFiles  PSScheduledJob  Completed  True        localhost   Get-ChildI...
16   RemoveLogFiles  PSScheduledJob  Completed  True        localhost   Get-ChildI...
19   RemoveLogFiles  PSScheduledJob  Completed  True        localhost   Get-ChildI...
22   Job22           BackgroundJob   Running    True        localhost   Get-Pro...
```

```
24   Job24           BackgroundJob   Running    True        localhost   Get-Pro...
26   Job26           BackgroundJob   Completed  True        localhost   Get-Pro...
28   CreateFile      PSScheduledJob  Completed  True        localhost   New-Ite...
29   CreateFile      PSScheduledJob  Completed  True        localhost   New-Ite...
```

Figure 22: The results or `Get-Job` shows two types of jobs, defined by the `PSJobTypeName` property.

Retrieving results of scheduled jobs is done the same way as with background jobs. Due to the nature of scheduled jobs, their results persist through PowerShell sessions, allowing results to be received whenever the user needs to examine the results. The results of scheduled jobs are saved on disk (Microsoft 2017u). As in background jobs, the results can be received only once if the `-Keep` switch parameter is not used. However, the results of scheduled job instances can be received again in another PowerShell session. The results on disk will only be removed if the job instance is removed with `Remove-Job` or if the scheduled job definition is deleted with `Unregister-ScheduledJob` (Jones, Hicks & Siddaway 2015, 192).

3.6.3 Utilizing scheduled automation

Using scheduled jobs can automate many tasks entirely by removing or minimizing user interaction. Schedules can remove the need for somebody to take care of a task. Just to find the time to complete a recurring task can take surprisingly high amounts of time. However, when tasks run on schedule, they still need to be monitored, even though the means are different from the user being present as the task runs. Monitoring can be implemented either by logging from within tasks or reviewing results of completed scheduled jobs by using the `Receive-Job` cmdlet. Manual monitoring should be kept to a minimum, so no redundant work is done. In many cases notifications or alerts can be implemented to tasks to point out the need for user action to certain individuals or teams.

As scheduled jobs do not limit management to jobs created in the current session, scheduled jobs could be used for basic centralization of job management. Even tasks that do not require schedules or triggers could be registered as scheduled

jobs, simply to have them available from any session. Tasks defined as scheduled jobs are pre-configured and can be executed more efficiently than creating the same background job multiple times. Job options, triggers and the job definition can be separately controlled, without a need to recreate the entire task when single component needs changes. The only problem in this is that the management of scheduled jobs is not centralized if all jobs are not managed from the same local computer. This limitation can be partially bypassed with PowerShell's remoting features, which, however, does partly reduce efficiency in managing the tasks.

The use cases of scheduled jobs include administrative tasks, for example removing logs, gathering information about servers or testing service availability, the main requirement being that the task could be run periodically. In larger environments, many tasks related to these use cases are not done simply because they would require too much time or organizing. Tasks like these best suited for scheduled jobs. Although the automation implementations will take time and effort, the invested time should be returned over time.

Due to the PSScheduledJob module's reliance on Windows Task Scheduler, the module is not available core PowerShell Core. On operating systems other than Windows, other scheduling methods, such as cron on unix systems, can be used. Many IT automation platforms also offer job scheduling and integrations into PowerShell or other command line tools. For example, Puppet (PowerShell module) and Ansible offer such integrations (Puppet 2018, Ansible 2018).

3.7 Workflows

A PowerShell workflow, available in PowerShell versions 3 - 5, is an advanced way of defining a function in PowerShell (Jones, Hicks & Siddaway 2015, 377). Similarly to functions, workflows are defined by using a specific syntax in PowerShell's scripting language. Workflows differ from functions by offering features, which allow users to better control the processing flow of a task, for example allowing parallel or sequential processing, recovery from failures and pausing or

resuming execution. PowerShell workflows follow the specifications of Windows Workflow Foundation (WWF), a core component of the .NET Framework (Microsoft 2018k). Launching a workflow from PowerShell first translates the PowerShell workflow definition into WWF-supported XAML (Extensible Application Markup Language), which is then executed by the WWF engine (Microsoft 2013b).

Workflows are most suitable for tasks that either take are long running or require steps to be executed in a certain order. Such tasks often exist in processes, which leverage a large number of remote computers, which leads to higher processing times and increased risk of failure. While the biggest gain is in these large environments, some features, especially parallelism, can be beneficial in minimizing runtimes in specific use cases.

3.7.1 Defining workflows

Workflows are defined with a set of keywords in PowerShell's scripting language. Workflows are considered as valid PowerShell code, they can be defined in script files and modules to be used as commands. As with functions, parameters can be defined to transfer data to the workflow. A workflow is defined with the `workflow` keyword with curly brackets, in which all the functionality is defined. An example of a simple workflow is presented in figure 9.

```
workflow Get-OSVersion
{
    (Get-CimInstance Win32_OperatingSystem).Version
}
```

Figure 23: The `Get-OSVersion` workflow outputs the operating system version.

Commands used inside workflows are called activities, which are the building blocks of workflows. Some activities change the execution behaviour of activities nested under them. These special activities, or blocks, are defined with the following keywords: `Sequence`, `Parallel`, `ForEach -Parallel` and `InlineScript`. Ac-

tivities inside a sequence blocks are executed sequentially, waiting for each command to finish until moving on the next one. Sequential execution is the default behaviour of workflows, so it only needs to be specified inside parallel blocks to specify sets of activities to be run in parallel (Microsoft 2017v). The parallel block runs activities at the same time. Although parallelism generally reduces the execution time, the workflow author needs to make sure the activities are not dependent on other activities' completion. When activities are run in parallel, there is no guarantee of which activity will complete first. The ForEach -Parallel block can be used in workflows to run activities simultaneously for each item in a collection. (Microsoft 2017w.)

3.7.2 Limitations in workflows

The translation of PowerShell commands into workflow activities creates some limitations, which need to be considered when designing and implementing workflows. Most cmdlets available natively in PowerShell work in workflows, although the way the work may change a bit. For example, some activities defined are executed on the local computer, even if the workflow is executed on a remote computer. There are also some changes in how objects and variables work inside workflows, because the workflow needs to be able to persist its state to enable pausing and resuming. Objects are deserialized to support persistence and remoting. The deserialization of objects removes the methods from objects, only the property values are saved. This limitation can be avoided with the InlineScript block which is used similarly to the sequence and parallel blocks. InlineScript is a piece of PowerShell script, executed by a local PowerShell session instead of the workflow engine. As the script is executed in PowerShell, limitations related to object deserialization in WWF can be bypassed. (Siddaway 2013a.)

Variables defined in a workflow, outside the scope of an InlineScript, are not automatically visible to InlineScripts. Workflow variables can be accessed from InlineScripts with a special syntax: `$using:<variable>`. Similarly, accessing variables from a higher-level scope from nested parallel or sequence blocks is possible with the `$workflow:<variable>` syntax. The value of a workflow variable cannot

be changed directly from inside an InlineScript, even with the `$using:` syntax, but workflow variable values can be set with the `$workflow:` syntax. The output of the InlineScript must be assigned to the variable in the workflow (`$result` in figure 24) (Microsoft 2017x). An example of this behaviour is shown in figure 24.

Figure 24 presents an example of using the InlineScript block. The `Find-ProcessVendor` workflow looks for a string in the target computer's processor name. The string is specified with the `-SearchString` parameter. Both the search string and processor names need to be converted to upper case to allow case-insensitive functionality. As the methods are not available inside the workflow, InlineScript is used to execute the comparison. The InlineScript returns a boolean value of `$True` or `$False`, depending on if the `$SearchString` is contained in the processor name. If the `Contains()` method returns `$True`, the processor information is returned.

```
# Find-ProcessorVendor.ps1
workflow Find-ProcessorVendor
{
    Param(
        [string]$SearchString
    )

    $processorInfo = Get-WmiObject win32_processor
    $result = InlineScript
    {
        $vendorName = ($using:searchString).ToUpper()
        ($using:processorInfo).Name.ToUpper().Contains($vendorName)
    }
    if ($result)
    {
        $processorInfo
    }
}
```

Figure 24: An example of a workflow defined in a script file. The workflow looks for processors containing a specific string.

3.7.3 Executing workflows

Workflows are executed with the same syntax as any cmdlet. As workflows are typically run in large environments on a lot of computers, support for remote execution is built into PowerShell workflows. All workflows support the `-PSComputerName` parameter, allowing the workflow to be run on one or more remote computers. By default, the workflow is run on current user. By using the `-PSCredential` parameter, another user account can be specified. (Microsoft 2017y.) The example shown in figure 25 executes the `Find-ProcessorVendor` workflow on two target computers, PowerShell automatically adds the `PSComputerName` property to the output to make it easy to identify where the results came from. Workflows are executed on all target computer in parallel, so the order of resulting output may vary.

```
PS C:\> Find-ProcessorVendor -SearchString "intel" -PSComputerName wserver5,wserver6


Caption           : Intel64 Family 6 Model 158 Stepping 9
DeviceID          : CPU0
Manufacturer      : GenuineIntel
MaxClockSpeed     : 4200
Name              : Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
SocketDesignation :
PSComputerName    : wserver6

Caption           : Intel64 Family 6 Model 158 Stepping 9
DeviceID          : CPU0
Manufacturer      : GenuineIntel
MaxClockSpeed     : 4200
Name              : Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
SocketDesignation :
PSComputerName    : wserver5
```

Figure 25: The `Find-ProcessorVendor` (As presented in figure 24) workflow is used to find computers with Intel processors.

The ability to stop, suspend, and resume a workflow execution is made possible by specifying persistence checkpoints, which save the state and data of the workflow execution so far on disk. The main benefit of using checkpoints is not having to restart a workflow from the beginning when its execution is interrupted by job suspension, a runtime error or loss of remote connectivity. However, the time taken to save the workflow execution state should be considered, only placing

checkpoints after critical actions. If during execution an error happens, or the workflow is suspended manually, execution can be resumed from the last persistence checkpoint. (Jones, Hicks & Siddaway 2015, 379-381.)

Persistence checkpoints can be created either manually or by specifying a behaviour for automatically creating them. Checkpoints can be specified manually in workflows with the `CheckPoint-Workflow` activity, which takes no parameters. Another way to manually add a checkpoint is to use the `-PSPersist` common parameter on any activity. The `-PSPersist` parameter creates a checkpoint after the activity execution has completed. The `-PSPersist` parameter can also be used on the workflow, affecting all activities in the workflow. By default, checkpoints are only added to the beginning and the end of a workflow. If `-PSPersist $True` is specified, checkpoints are also added after each activity. `-PSPersist $False` states that no checkpoints are added automatically, although checkpoints can still be manually placed. Automatic checkpointing can also be controlled from within a workflow with the `$PSPersistPreference` boolean variable, which places a checkpoint after every executed workflow as long as its value is `$True`. The variable can be used to dynamically control checkpoints in the workflow logic. (Microsoft 2017y.) The value is set to `$False` by default.

3.7.4 Workflows as jobs

Workflows can be managed as PowerShell jobs by using the `-AsJob` common parameter when calling the workflow. Optionally, the `-JobName` can also be used to define a customized name for the job. With these parameters, the workflow is executed as any other job, enabling execution management and monitoring as well as receiving output from workflow execution. Workflows started as jobs can be identified with the `PSJobTypeName` property of the job, `PSWorkflowJob` being the category for workflow jobs. In addition to having access to basic job functionality, workflows enable usage of some special cmdlets. The `Suspend-Job` cmdlet can be used to pause a workflow job. When a workflow job is suspended, the execution will pause at the next persistence checkpoint, so no data loss will occur. The

workflow job will enter the Suspended state, partial results up to the latest checkpoint can be received normally. Workflow jobs can also be suspended from the workflow with the `Suspend-Workflow` activity, which moves the workflow job to Suspended state. `Suspend-Workflow` will create a new job if the `-AsJob` is not specified. This behaviour is shown in figure 26. Jobs in Suspended state can be resumed from the latest saved checkpoint with the `Resume-Job` cmdlet. When a workflow job is resumed, its execution state and data is reconstructed from a saved persistence checkpoint.

```
# Test-Suspend.ps1
workflow Test-Suspend {
     Write-Output "Setting `$a value to 123"
     $a = "123"
     Write-Output "Suspending"
     Suspend-Workflow
     Write-Output "Sleeping for 30 seconds"
     Start-Sleep -Seconds 30
     Write-Output "Writing output"
     $a
}


---


PS C:\> Test-Suspend
Setting $a value to 123
Suspending

Id   Name    PSJobTypeName   State       HasMoreData   Location    Command
--   ----    -------------   -----       -----------   --------    -------
83   Job83   PSWorkflowJob   Suspended   True          localhost   Test-Suspend


PS C:\> Resume-Job 83

Id   Name    PSJobTypeName   State     HasMoreData   Location    Command
--   ----    -------------   -----     -----------   --------    -------
83   Job83   PSWorkflowJob   Running   True          localhost   Test-Suspend

PS C:\> Receive-Job 83
Sleeping for 30 seconds
Writing output
123
```

Figure 26: A demonstration of suspending a workflow.

In figure 26, the `Test-Suspend` workflow contains the `Suspend-Workflow` activity, which suspends the workflow and creates a persistence checkpoint (Microsoft 2017z). As soon as the workflow execution reaches the `Suspend-Workflow` activity, the suspended job information is output. The execution is resumed then with

`Resume-Job`. After the execution has finished, the remaining output is received with `Receive-Job`.

3.7.5 Utilizing workflows

Workflows are able to recover from interruptions such as unplanned computer restarts or connectivity failures automatically by utilizing persistence checkpoints. When an interruption occurs, execution is resumed from the last persisted checkpoint as soon as the computer is available. (Microsoft 2016b.) This functionality allows running workflows on considerably higher amounts of remote computers, where availability is not granted. Time is saved when multiple individual remotely executed workflows do not have to be audited or debugged for errors that could be avoided with workflows.

Workflows can also be used to handle manual computer restarts inside tasks, which is a fairly common need in administrative tasks. Persisting data through a computer restart can be very difficult from a single script. By using workflow, task execution can continue after restarting the computer by saving the execution state on disk beforehand. A restart can be initiated with the `Restart-Computer -Wait` command, the `-Wait` switch parameter ensures the computer is running before attempting to continue execution. (Siddaway 2013b.)

Workflows are an alternative way to implement tasks in addition to functions and scripts. The benefits of using workflows are not significant until the complexity of an automation reaches a certain point. The specific use cases for workflows can be hard to identify, but certain aspects should be considered. Long execution times and running a task on tens or hundreds of computers will lead to a more error prone automation, which would be difficult to maintain without workflows.

The implementation of workflows in PowerShell itself is impressive. Even though the underlying technology is entirely separate from PowerShell, the implementation resembles PowerShell syntax at first glance. Of course the workflow implementations do require the user to get used to new rules and limitations, possibly

creating some confusion. However, once a workflow is implemented, it is used like any other cmdlet or function. Workflows can be run as jobs, on-demand or scheduled, or used as part of a bigger automation.

4 In practice

The target areas for automating tasks with PowerShell include data manipulation and management, integration, maintenance, monitoring, configuration, provisioning and application deployments. Just the native functionality of PowerShell can cover many of the use cases in these areas. Even if PowerShell's capability is extended, the task automation features introduced in chapter 3 are able to provide a supporting framework for automating specific tasks.

This chapter focuses on practical use cases that are able to display PowerShell's native automation features diversely. All examples describe the use case briefly and contain the steps to implement the automation. The examples are not meant to be complete, production-ready automations, but rather be sufficient to demonstrate the automation capability of PowerShell, with a focus on the features and concepts introduced in this thesis.

4.1 Archiving files

Many systems and applications produce log files, often split into smaller files, which contain log data from a period. Automating the periodic archiving of log files saves disk space. The implemented example for this use case is a script, which can be configured with parameters. The script searches for files older than the time specified with the `-DateFilter` parameter. If no files are found, the execution will end, producing no output. If files are found, they are archived using the `Compress-Archive` cmdlet, which compresses the files into a zip archive. If the files are successfully compressed, they are removed, so only the compressed zip file remains.

The target folder is specified with the `-Path` parameter. The `-Destination` parameter is used to define a path where the zip file is saved. Optionally, the `-LastWrite` switch parameter can be included, to use the `LastWriteTime` property instead of `CreationTime` when searching for files older than the `-DateFilter` parameter value. After the files are compressed, the script writes the resulting zip file path as output, to allow users to easily manipulate the resulting file. The script is defined as follows:

```
# Archive-Files.ps1

[CmdletBinding()]
Param(
    [Parameter(Mandatory=$true)]
    [string]$Path,
    [Parameter(Mandatory=$true)]
    [string]$Destination,
    [Parameter(Mandatory=$true)]
    [DateTime]$DateFilter,
    [switch]$LastWrite
)

# Use LastWriteTime for filtering if -LastWrite is used
$filterProperty = if($LastWrite) { "LastWriteTime" } else { "CreationTime" }

# Look for files older than $DateFilter in $Directory
Write-Verbose "Looking for files older than $($DateFilter.ToString("yyyy-MM-dd")) in
$Path"
$files = Get-ChildItem -Path $Path -File |
    Where-Object -Property $filterProperty -lt $DateFilter |
    Select-Object -ExpandProperty "FullName"
if($files.Length -gt 0) {
    try {
        Write-Verbose "Archiving $($files.Count) files"
        Compress-Archive -Path $files -DestinationPath $Destination -ErrorAction
"Stop"
        Write-Verbose "Removing files"
        $files | Remove-Item
        Write-Output (Get-ChildItem $Destination | Select-Object -First 1)
    } catch
    {
        Write-Error $_
    }
} else
{
    Write-Verbose "No files found"
}
```

Because the archiving is done periodically, the script can be scheduled using scheduled jobs. In this example, the job triggered every 2 weeks on Sunday at 05:00. The commands used to schedule the job are as follows:

```
$jobTrigger = New-Jobtrigger -Weekly -WeeksInterval 2 -At 05:00 -DaysOfWeek Sunday
$commands = {
    $currentDate = Get-Date -Format "yyyy-MM-dd"
    $dateFilter = (Get-Date).AddDays(-14)
    C:\powershell\Archive-Files.ps1 -Path C:\powershell\log -Destination
    "c:\logarchive\$currentDate.zip" -DateFilter $dateFilter -
    LastWrite -Verbose *>> "C:\log\archivelog_$currentDate.log"
}
Register-ScheduledJob -Name "ArchiveLogFiles" -ScriptBlock $commands -Trigger
$JobTrigger
```

The scheduled job is registered using a script block instead of referencing the script file directly to allow some additional commands adding timestamps dynamically in relation to current time. The `$dateFilter` variable is set to a date exactly two weeks before the current time. The resulting zip file is also named with a timestamp. Every two weeks a zip file will be created, containing the log files from past two weeks.

The individually listed jobs' output can be monitored by using the `Get-Job` and `Receive-Job` cmdlets. The output produced by `Write-Verbose` is also visible, because the `-Verbose` switch parameter is specified when the script is called. Although the produced output is not very extensive in this example, the redirection operator `*>>` is used to write the output to a log file automatically.

The scheduled job features offer a nice way to manage the job. Due to the modular implementation of the script, it can be used in multiple use cases. New triggers can be added to tighten the periodic archiving, or new jobs registered to implement archiving of files, all using the same script file.

4.2 Collecting information about computers

This example focuses on collecting essential information about computers connected to the same Active Directory (AD) domain. The data is retrieved by using the PowerShell module for AD management in combination with other cmdlets, which are executed on remote computers. Retrieving information about computers useful in many tasks related to maintenance and monitoring. In this example, the retrieved information includes the target computer's name, operating system,

operating system version, IP address, default gateway and DNS (Domain Name System) server address.

The example implementation does not require any preconfigured information about the target computers. Computers connected to the same AD domain can be retrieved with the `Get-ADComputer` cmdlet, which is a part of the Active Directory module for Windows PowerShell. The Active Directory module is available on Windows Server systems as an installable Windows feature. On Desktop versions of Windows, an installation of the Remote Server Administration Tools (RSAT) is required. Operating system information and the IP address of a computer are included in the output of `Get-ADComputer`. The rest of the information is retrieved with `Get-NetIPConfiguration`, which is invoked on all target machines by using remoting features. As the `Get-NetIPConfiguration` cmdlet requires the target computer to be available for remoting, the results in this example are limited to currently running computers. The script implemented for this example is as follows:

```
[CmdletBinding()]
Param(
    $ComputerFilter = "*"
)

Write-Verbose "Getting AD computers"
$computers = Get-ADComputer -Property * -Filter $ComputerFilter
Write-Verbose "Total computers: $($computers.Count)"
$computers[0].IPv4Address -Match "\d+\.\d+\." | Out-Null
$networkFilter = "$($matches[0])*"
Write-Verbose "Detected network: $networkFilter"
$computerNames = $computers | Select-Object -ExpandProperty Name

$commands = {
    Param(
        $Filter
    )
    $netConfig = Get-NetIPConfiguration | Where-Object {
        $_.IPv4Address.IPAddress -like $Filter
    }
    $ipAddress = $netConfig.IPv4Address.IPAddress
    $defaultGateWay = $netConfig.IPv4DefaultGateWay.NextHop
    $DNSServer = $netConfig.DNSServer | Select-Object -ExpandProperty ServerAddresses
    [PSCustomObject]@{
        IP = $ipAddress
        DefaultGateway = $defaultGateWay
        DNSServer = $DNSServer
    }
}

Write-Verbose "Invoking remote commands"
$results = Invoke-Command -ScriptBlock $commands -ArgumentList $networkFilter -
ComputerName $computerNames -ErrorAction SilentlyContinue
```

```
ForEach($result in $results) {
    $adInfo = $computers | Where-Object -Property Name -eq $result.PSComputerName
    [PSCustomObject]@{
        ComputerName = $adInfo.Name
        OSName = $adInfo.OperatingSystem
        OSVersion = $adInfo.OperatingSystemVersion
        IP = $result.IP
        DefaultGateway = $result.defaultGateWay
        DNSServer = $result.DNSServer
    }
}
```

The script contains only one parameter, which can optionally be used to filter the computers returned by `Get-ADComputer` by name. After the computers are retrieved with `Get-ADComputer`, the script detects the network used by the domain to identify correct network adapters for DNS and default gateway addresses. The output of the script is shown below.

```
PS C:\> C:\powershell\Get-ADComputerInformation -Verbose
VERBOSE: Getting AD computers
VERBOSE: Total computers: 3
VERBOSE: Detected network: 192.168.*
VERBOSE: Invoking remote commands


ComputerName   : WSERVER7
OSName         : Windows Server 2016 Datacenter
OSVersion      : 10.0 (14393)
IP             : 192.168.1.107
DefaultGateway : 192.168.1.1
DNSServer      : 192.168.1.101

ComputerName   : WSERVER5
OSName         : Windows Server 2016 Datacenter
OSVersion      : 10.0 (14393)
IP             : 192.168.1.101
DefaultGateway : 192.168.1.1
DNSServer      : 127.0.0.1

ComputerName   : WSERVER6
OSName         : Windows Server 2016 Datacenter
OSVersion      : 10.0 (14393)
IP             : 192.168.1.103
DefaultGateway : 192.168.1.1
DNSServer      : 192.168.1.101
```

The output could be used for various purposes. Redirection could be used to log these results to file, while in many cases the output might be hard to read from a log file, especially if the number of computers is high. The `Export-Csv` cmdlet could be used to write just the success output to a csv file. The data could then be imported to any PowerShell session with `Import-Csv`. While in this example

the information retrieved by using `Invoke-Command` is limited to network configuration, the script could be extended to include virtually any information retrievable with PowerShell commands. For example, retrieving version information of installed applications could be vital in finding security vulnerabilities within an IT environment. If the commands executed with `Invoke-Command` would increase the execution time considerably, it might be useful to schedule the task to run periodically, saving the results to a file for easy access.

4.3 Adding computers to a domain

In this example, a workflow is implemented to join computers to a domain by using a workflow. The implementation also configures the DNS server address on the target computer and tests connectivity to the DNS server after the domain has been joined. Workflows allow execution to continue past the required restart, allowing all actions to run without user interaction. The implemented workflow is as follows:

```
workflow AddComputerToDomain {
    Param(
        [Parameter(Mandatory=$True)]
        [string]$DomainName,
        [Parameter(Mandatory=$True)]
        [PSCredential]$Credential,
        [Parameter(Mandatory=$True)]
        [string]$DNSServer
    )
    Write-Verbose "Setting DNS server address: $DNSServer"
    $netAdapter = InlineScript {
        $VerbosePreference = "SilentlyContinue"
        $using:DNSServer -Match "\d+\.\d+\." | Out-Null
        $networkFilter = "$($matches[0])*"
        $netAdapter = Get-NetIPConfiguration |
            Where-Object { $_.IPv4Address.IPAddress -like $networkFilter }
        $netAdapter
    }
    Set-DNSClientServerAddress -InterfaceIndex $netAdapter.InterfaceIndex -
ServerAddresses $DNSServer
    Write-Verbose "Adding computer to domain"
    $localCredential = InlineScript {
        $localUserName = "$($env:ComputerName)\$($using:Credential.UserName)"
        New-Object System.Management.Automation.PSCredential (
            $localUserName,
            $using:Credential.Password
        )
    }
    Add-Computer -DomainName "wlab" -Credential $localCredential
    Write-Verbose "Restarting computer"
    Restart-Computer -Wait -Force
    Write-Verbose "Testing connection to DNS server"
```

```
    $testResult = (Test-NetConnection -ComputerName $DNSServer)
    if ($testResult.PingSucceeded) {
        $connectionStatus = "OK"
    } else {
        $connectionStatus = "FAIL"
    }
    [PSCustomObject]@{
        "DomainName" = $DomainName
        "DNSConnectionStatus" = $connectionStatus
    }
}
```

First the workflow configures the DNS server by using the `Set-DNSClientServerAddress` cmdlet. After that, the computer is added to the domain with `Add-Computer`, which requires a local credential with sufficient permissions. Then the computer is restarted using the `Restart-Computer` cmdlet. After the restart completes, the workflow ensures connectivity to the DNS server with the `Test-NetConnection` cmdlet. The output includes the `DNSConnectionStatus` property, indicating the result of the connectivity test. InlineScript blocks are used for some parts to enable usage of features, that are not allowed in workflows, such as regular expressions, which are used to identify the current network adapter for retrieving the DNS configuration.

With the remoting functionality built into workflows, the workflow can be invoked on multiple target computers in parallel. The workflow expects that the credential provided with the `$Credential` parameter is valid on all target computers. The workflow could be invoked with different connection credentials for each target computer, but in this example the same connection credentials are valid. The workflow is used to add two computers into the “wlab” domain with the following commands:

```
PS C:\powershell> Start-Transcript -Path -Append "C:\log\$(Get-Date -Format "yyyy-MM-
dd_HH-mm-ss").log"
Transcript started, output file is C:\log\2018-05-19_19-33-01.log

PS C:\powershell> $credential = Get-Credential

PS C:\powershell> AddComputerToDomain -DomainName "wlab" -PSComputerName
"192.168.1.108","192.168.1.103" -DNSServerAddress "192.168.1.101" -PSCredential
$credential -Credential $credential -Erroraction Stop -Verbose

VERBOSE: [192.168.1.108]:Setting DNS server address: 192.168.1.101
VERBOSE: [192.168.1.103]:Setting DNS server address: 192.168.1.101
VERBOSE: [192.168.1.103]:Adding computer to domain
VERBOSE: [192.168.1.103]:Performing the operation "Join in domain 'wlab'" on target
"WSERVER6".
```

```
VERBOSE: [192.168.1.108]:Adding computer to domain
VERBOSE: [192.168.1.108]:Performing the operation "Join in domain 'wlab'" on target
"WServer8".
WARNING: [192.168.1.103]:The changes will take effect after you restart the computer
WSERVER6.
VERBOSE: [192.168.1.103]:Restarting computer
VERBOSE: [192.168.1.103]:Performing the operation "Enable the Remote shutdown access
rights and restart the computer." on target "192.168.1.103".
VERBOSE: [192.168.1.103]:Testing connection to DNS server


DomainName            : wlab
DNSConnectionStatus   : OK
PSComputerName        : 192.168.1.103
PSSourceJobInstanceId : fdd789a7-c06c-47ec-974d-f1c4506cf5bc

WARNING: [192.168.1.108]:The changes will take effect after you restart the computer
WServer8.
VERBOSE: [192.168.1.108]:Restarting computer
VERBOSE: [192.168.1.108]:Performing the operation "Enable the Remote shutdown access
rights and restart the computer." on target "192.168.1.108".
VERBOSE: [192.168.1.108]:Testing connection to DNS server


DomainName            : wlab
DNSConnectionStatus   : OK
PSComputerName        : 192.168.1.108
PSSourceJobInstanceId : 161ceefa-34c6-44dc-be7b-e378f4aea7b5


PS C:\powershell> Stop-Transcript
Transcript stopped, output file is C:\log\2018-05-19_19-33-01.log
```

The verbose output is displayed in whatever order the remote execution progresses, the IP address of the target computer is automatically added to verbose and warning messages. PowerShell's transcript feature is used to log the output to a file.

5 Conclusion

The goal of this thesis was to research and demonstrate PowerShell's capability as a task automation platform. Features related to implementation, execution, management and monitoring of tasks were examined to define the capability. These features were then taken into practice by demonstrating their usage with practical use cases. This chapter aims to conclude the thesis by evaluating the researched features and task automation capability in general.

Learning PowerShell and its many diverse conventions seemed to become easier over time. The conventions and rules often differ from traditional command line tools or programming languages, but start to make sense as you get deeper and deeper into the learning process. The consistency in PowerShell's native functions makes it very easy to learn how to use cmdlets or features with just the shell by using the built-in documentation. When you have worked with PowerShell for a while, the transition into building scripts and modules seems intuitive, requiring minimal effort in identifying good conventions. Just by using PowerShell, you probably know how a function or a script should be named or how it should produce output.

PowerShell presents many different ways to implement automations. The underlying concept, however, remains the same. All tasks are implemented by defining valid PowerShell code consisting of commands in the object pipeline, optionally wrapped in PowerShell's scripting language. Implementations are made reusable by placing them in code wrappers: script blocks, script files, functions, workflows and modules. All of these wrappers function differently, some can only exist in a session, some can be saved to a file and imported into a PowerShell session. The wrappers offer a lot of flexibility, in some cases too much, as it can be hard to know which one to use. The time saved by automating a task should always be considered when choosing the implementation method, in many cases it might simply be faster to quickly implement a script, not putting too much thought on its generic design, which might be required to create a reusable module. However, if the task can be implemented in a generic way, allowing other specific use cases to utilize the implementation, it may be worth to invest the time to make a library of functions or workflows and distribute it as modules.

The researched features related to the supporting framework for tasks are extremely modular and rarely seem dependent on each other. The modular parts of this framework researched in this thesis include remoting, logging, background jobs and scheduled jobs. Each modular part of the framework can be attached to a task, making the type or complexity of a task irrelevant. Task implementations ranging from singular commands to long running complex workflows can have the exact same supporting features, such as logging or scheduling. This concept

supports the generic structure of a task, leaving the usage of these features up to the user automating a specific task. As with PowerShell in general, the modular design of the task automation framework can be broken by not following conventions and certain principles in task implementations. The general rule is that a generic structure of a task, such as a function or a workflow, should never contain usage of the supporting framework. However, when tasks are scheduled, the logic specific to the use case should be placed in a wrapping, higher level script, which ensures the specific automation can be executed and distributed easily. For example, logging can always be left up to the user, but if a task is scheduled, the log file names can be named with timestamps by defining some surrounding logic.

All in all the researched features cover the most essential requirements of a task automation framework. Tasks can be created with various methods, depending on the use case and complexity of a task. Any task can be managed as a background job or scheduled with scheduled jobs. Monitoring is supported by the logging features as well as jobs. Remoting brings options for executing tasks without physical access to computers. All of these parts work very well on their own. The biggest downside is that some of these features feel partly disconnected from PowerShell due to their dependency on some external technology. Scheduled jobs' reliance on the Windows Task Scheduler feels like a cheap approach, a built-in integration into another Windows component, which is relatively old. Workflows are another example of the same concept, an external technology integrated into PowerShell. Both of these satisfy an essential need, with limited equivalent features available. Both are also not available in PowerShell Core, so similar functionality must be replaced with extensions or external tools.

In addition to providing their main functions, background and scheduled jobs support basic management of tasks relatively well. While the main use of background jobs is in working with the interactive shell, scheduled jobs can be used as a kind of a task manager for independent automations. The functionality in itself is a bit too basic and requires some effort in task definitions make automations independent. Features related to monitoring, such as logging and error alerts, must be implemented within the task.

The .NET Framework brings an extremely wide range of capability to PowerShell. PowerShell's scripting language is not as extensive as other .NET programming languages, but will offer more than enough features for shell-based automations. Extensions can be implemented with PowerShell code or a .NET programming to integrate into any existing system.

As a task automation platform, PowerShell offers the most essential functionality. For enterprise-scale automations, however, some work would be required to have all the required functionality available. If just the native functionality is used, some steps in automations will inevitably be repeated. External software can run PowerShell, so many of these functionalities could be implemented elsewhere. However, PowerShell does hold a strong ground as an interactive shell environment, especially in automating tasks performed on Windows systems, where it is in many cases irreplaceable.

This thesis demonstrated the researched features with practical examples to illustrate the changes task automation creates. The automation of a task will create other smaller tasks separate from the actual performing of a technical task. A person can remember what they did and describe the process. An automation must also be able do that, but the methods change drastically. These changes can be hard to recognize beforehand, but they will emerge later. Just following simple technical examples will not emphasize the real-world effects of automation well enough.

PowerShell documents its own functionality very effectively with the integrated help topics, or manual pages as they are called in many shells. Many of these help topics were used in the process of creating this thesis in researching and learning PowerShell's features and concepts. Especially the "*about_feature*" topics, which can be viewed by using the `Get-Help` cmdlet seem to provide extremely extensive information about PowerShell's features and concepts, functioning as an official documentation built into the shell. Many web-based representations of these help topics are referenced in this thesis because of this.

I have learned a lot about the concepts of command line tools during the process of creating this thesis. Even though some concepts of PowerShell, like naming conventions or the object-oriented approach, differ from most command lines, the basic functionality is the same. Experience of using a command line tool should be useful in the future, providing ways to automate tasks in IT. I feel like I have a new multi-tool in my pocket to use, applicable for a wide variety of problems. As in my case, I think any professional working in the IT field would find tasks that could easily be automated in their work, if they just knew how easy PowerShell is to use.

As a shell, PowerShell is modern when compared to text-based shells. Many steps have been taken to innovate and extend a shell's capability. I expect this will continue with PowerShell Core, taking the shell into a broad range of new operating systems and applications.

References

Ansible. 2018. Integration: Ansible and Windows. Red Hat. https://www.ansible.com/integrations/infrastructure/windows. 29.4.2018.

Blender, J. 2014. Understanding Streams, Redirection, and Write-Host in PowerShell. Hey, Scripting Guy! Blog. 30.3.2014. https://blogs.technet.microsoft.com/heyscriptingguy/2014/03/30/understanding-streams-redirection-and-write-host-in-powershell/.

Jones, D. 2012. Windows PowerShell: Defining Parameters. https://technet.microsoft.com/en-us/library/jj554301.aspx. 8.5.2018.

Jones, D., Hicks, J. & Siddaway, R. 2015. PowerShell In Depth Second Edition. New York. Manning.

MacKechnie, N. 2005. msh: Microsoft Command Shell (Codename: Monad) Beta 2 Refresh. PowerShell Team Blog. 2.11.2005. https://blogs.msdn.microsoft.com/nickmac/2005/11/02/msh-microsoft-command-shell-codename-monad-beta-2-refresh/. 28.2.2018.

Microsoft. 2006a. Windows PowerShell (Monad) Has Arrived. PowerShell Team Blog. 25.4.2006. https://blogs.msdn.microsoft.com/powershell/2006/04/25/windows-powershell-monad-has-arrived/. 28.02.2018.

Microsoft. 2006b. It's A Wrap! Windows PowerShell 1.0 Released!. PowerShell Team Blog. 14.11.2006.

https://blogs.msdn.microsoft.com/powershell/2006/11/14/its-a-wrap-windows-powershell-1-0-released/. 28.02.2018.

Microsoft. 2008. PowerShell will be installed by default on Windows Server 08 R2 (WS08R2) and Windows 7 (W7)!. PowerShell Team Blog. 28.10.2008. https://blogs.msdn.microsoft.com/powershell/2008/10/28/powershell-will-be-installed-by-default-on-windows-server-08-r2-ws08r2-and-windows-7-w7/. 28.2.2018.

Microsoft. 2009. Windows Management Framework is here!. PowerShell Team Blog. 27.10.2009. https://blogs.msdn.microsoft.com/powershell/2009/10/27/windows-management-framework-is-here/. 28.2.2018.

Microsoft. 2012a. Windows Management Framework 3.0 Available for Download. PowerShell Team Blog. 17.9.2012. https://blogs.msdn.microsoft.com/powershell/2012/09/17/windows-management-framework-3-0-available-for-download/. 28.2.2018.

Microsoft. 2012b. Windows PowerShell: Defining parameters. https://technet.microsoft.com/en-us/library/jj554301.aspx. 10.5.2018.

Microsoft. 2013a. Windows Management Framework 4.0 is now available. PowerShell Team Blog. 24.10.2013. https://blogs.msdn.microsoft.com/powershell/2013/10/24/windows-management-framework-4-0-is-now-available/. 28.2.2018.

Microsoft. 2013b. Windows PowerShell: What is Windows PowerShell Workflow?. Microsoft. https://technet.microsoft.com/en-us/library/jj884462.aspx. 30.4.2018

Microsoft. 2016a. Windows Management Framework (WMF) 5.0 RTM packages has been republished. PowerShell Team Blog. 24.2.2016. https://blogs.msdn.microsoft.com/powershell/2013/10/24/windows-management-framework-4-0-is-now-available/. 28.2.2018.

Microsoft. 2016b. Restarting the Computer in a Workflow. Microsoft. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/jj574130(v%3dws.11). 6.5.2018.

Microsoft. 2017a. PowerShell Overview. Microsoft. https://docs.microsoft.com/en-sg/powershell/scripting/powershell-scripting. 25.02.2018.

Microsoft. 2017b. Getting Started with Windows PowerShell. Microsoft. https://docs.microsoft.com/en-us/powershell/scripting/getting-started/getting-started-with-windows-powershell. 25.2.2018.

Microsoft. 2017c. Windows Management Framework (Windows PowerShell 2.0, WinRM 2.0, and BITS 4.0). Microsoft. https://support.microsoft.com/en-us/help/968929/windows-management-framework-windows-powershell-2-0-winrm-2-0-and-bits. 28.2.2018

Microsoft. 2017d. Windows PowerShell System Requirements. Microsoft. https://docs.microsoft.com/en-us/powershell/scripting/setup/windows-powershell-system-requirements. 28.2.2018.

Microsoft. 2017e. Windows Management Framework 3.0. Microsoft. https://www.microsoft.com/en-us/download/details.aspx?id=34595. 28.2.2018.
Microsoft. 2017f. Windows Management Framework 4.0. Microsoft. https://www.microsoft.com/en-us/download/details.aspx?id=40855. 28.2.2018.
Microsoft. 2017g. Understanding Important Windows PowerShell Concepts. Microsoft. https://docs.microsoft.com/en-sg/powershell/scripting/getting-started/fundamental/understanding-important-windows-powershell-concepts. 1.3.2018
Microsoft. 2017h. Learning Windows PowerShell Names. Microsoft. https://docs.microsoft.com/en-sg/powershell/scripting/getting-started/fundamental/learning-windows-powershell-names. 27.2.2018
Microsoft. 2017i. Getting Detailed Help Information. Microsoft. https://docs.microsoft.com/en-us/powershell/scripting/getting-started/fundamental/getting-detailed-help-information. 27.2.2018.
Microsoft. 2017j. Understanding the Windows PowerShell Pipeline. Microsoft. https://docs.microsoft.com/en-us/powershell/scripting/getting-started/fundamental/understanding-the-windows-powershell-pipeline. 27.2.2018.
Microsoft. 2017k. Using Variables to Store Objects. Microsoft. https://docs.microsoft.com/en-us/powershell/scripting/getting-started/fundamental/using-variables-to-store-objects. 9.5.2018
Microsoft. 2017l. About Variables. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_variables. 9.5.2018
Microsoft. 2017m. About Operators. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_operators. 11.5.2018
Microsoft. 2017n. About Scripts. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scripts. 11.5.2018.
Microsoft. 2017o. About CommonParameters. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_commonparameters. 12.5.2018.
Microsoft. 2017p. Start-Transcript. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.host/start-transcript. 12.5.2018.
Microsoft. 2017q. About Remote Output. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_remote_output. 10.5.2018.
Microsoft. 2017r. About Jobs. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_jobs. 9.4.2018.
Microsoft. 2017s. About Remote Jobs. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_remote_jobs. 16.4.2018.

Microsoft. 2017t. About Scheduled Jobs. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psscheduledjob/about/about_scheduled_jobs. 18.4.2018.

Microsoft. 2017u. About Scheduled Jobs Advanced. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psscheduledjob/about/about_scheduled_jobs_advanced. 28.4.2018.

Microsoft. 2017v. About Sequence. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psworkflow/about/about_sequence. 1.5.2018.

Microsoft. 2017w. About Parallel. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psworkflow/about/about_parallel. 1.5.2018.

Microsoft. 2017x. About InlineScript. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psworkflow/about/about_inlinescript. 3.5.2018.

Microsoft. 2017y. About WorkflowCommonParameters. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psworkflow/about/about_workflowcommonparameters. 1.5.2018.

Microsoft. 2017z. About Suspend-Workflow. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psworkflow/about/about_suspend-workflow. 6.5.2018.

Microsoft. 2018a. What's New in PowerShell Core 6.0. Microsoft. https://docs.microsoft.com/en-us/powershell/scripting/whats-new/what-s-new-in-powershell-core-60. 6.3.2018.

Microsoft. 2018b. Understanding a Windows PowerShell Module. Microsoft. https://msdn.microsoft.com/en-us/library/dd878324(v=vs.85).aspx. 27.2.2018.

Microsoft. 2018c. How to Write a PowerShell Module Manifest. Microsoft. https://msdn.microsoft.com/en-us/library/dd878337(v=vs.85).aspx. 12.3.2018.

Microsoft. 2018d. Modules and Snap-ins. Microsoft. https://msdn.microsoft.com/en-us/library/dd878246(v=vs.85).aspx. 26.2.2018.

Microsoft. 2018e. Write-Verbose. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-verbose. 12.5.2018.

Microsoft. 2018f. Write-Debug. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-debug. 12.5.2018.

Microsoft. 2018g. About Preference Variables. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_preference_variables. 12.5.2018.

Microsoft. 2018h. About Remote Requirements. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_remote_requirements. 10.5.2018.

Microsoft. 2018i. Invoke-Command. Microsoft. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/invoke-command. 10.5.2018.

Microsoft. 2018j. Unregister-ScheduledJob. Microsoft. https://docs.microsoft.com/en-us/powershell/module/psscheduledjob/unregister-scheduledjob. 28.4.2018.

Microsoft. 2018k. Understanding Windows Workflow Foundation. Microsoft. https://msdn.microsoft.com/en-us/library/cc303703.aspx. 30.4.2018.

Puppet. 2018. puppetlabs-powershell. GitHub. https://github.com/puppetlabs/puppetlabs-powershell. 29.4.2018.

Siddaway, R. 2013a. PowerShell Workflows: Restrictions. Hey, Scripting Guy! Blog. 2.1.2013. https://blogs.technet.microsoft.com/heyscriptingguy/2013/01/02/powershell-workflows-restrictions/. 3.5.2018.

Siddaway, R. 2013b. PowerShell Workflows: Restarting the Computer. Hey, Scripting Guy! Blog. 23.1.2013. https://blogs.technet.microsoft.com/heyscriptingguy/2013/01/23/powershell-workflows-restarting-the-computer/. 6.5.2018.

Snover, J. 2016. PowerShell is open sourced and is available on Linux. Microsoft Azure Blog. 18.8.2016. https://azure.microsoft.com/en-us/blog/powershell-is-open-sourced-and-is-available-on-linux/. 28.2.2018.